

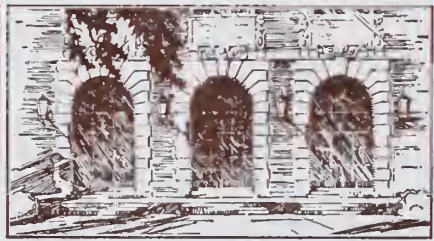
LIBRARY OF THE
UNIVERSITY OF ILLINOIS
AT URBANA-CHAMPAIGN

510.84

Il6r

no. 830-835

cop. 2





10.84
UIUCDCS-R-76-834

Math Lib

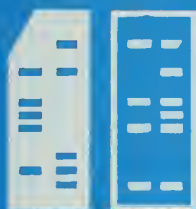
8

166
no. 834
20p2
IMPLEMENTATION OF THE LANGUAGE CLEOPATRA:
THE ANALYSIS PASS

by

Scott Harley Fisher

October 1976



DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

The Library of the

FEB 18 1977

University of Illinois
* Urbana-Champaign

LIBRARY OF L. URBANA CHAMPAIGN

UIUCDCS-R-76-834

IMPLEMENTATION OF THE LANGUAGE CLEOPATRA:
THE ANALYSIS PASS

BY

SCOTT HARLEY FISHER

B.S., University of Illinois, 1972

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1976

Urbana, Illinois

PROPERTY OF U. S. OF I. URBANA-CHAMPAIGN

Acknowledgement

As I write this acknowledgement and think back over the long months of work that went into this project, it is difficult to name the individuals who have influenced this work. To any deserving but unnamed individuals, I express my regrets for the omission.

I would like to thank my thesis advisor Dr. H. George Friedman, Jr. for suggesting this project, reading this thesis, and supplying the proper environment for the work contained herein.

When embarking on a new field, it is always beneficial to study the previous literature on the subject. In my case, the "previous literature" was my Director of Research, but more than that, my "previous literature" was embodied in an enthusiastic pedagogue and good friend--Dr. Axel T. Schreiner. Axel was tremendously helpful, and always willing to discuss difficult points. He also displayed

LIBRARY OF THE
OF THE
URBANA-CHAMPAIGN

confidence in my abilities and in the project even when confidence seemed out of order. It seems inadequate, but--thanks Axel.

I also extend a special thanks to my family. Although they do not understand the esoterica involved in the analysis pass of a compiler, they were always confident in and supportive of my work. It is, to a large part, through their efforts that I have attained this goal. For that, I sincerely thank them all.

Lastly, I wish to express my thanks to two fellow graduate students and good friends--John Modry and John Bowman--for helpful suggestions and technical advice on various aspects of this project.

It is to these people that I dedicate this thesis.

S. H. P.

LIBRARY OF L. URBANA-CHAMPAIN

Table of Contents

Chapter	Page
1.0 Introduction	1
2.0 Accessibility and Scope	3
2.1 The Configuration	4
2.2 Local vs. Global Scope	6
2.2.1 Global Scope	6
2.2.2 Local Scope	8
3.0 The Subset Language	9
3.1 Structure Blocks	10
3.2 Data Blocks	11
3.3 Routine Blocks	12
4.0 Implementation of the Subset	13
4.1 Implementation Rules for Structure Blocks ...	13
4.2 Implementation Rules for Data Blocks	16
4.3 Implementation Rules for Routine Blocks	18
4.4 Statements	19
5.0 Further Considerations	27
5.1 Pointer Variables	27
5.2 Presentation of Blocks for Compilation	30
6.0 Structure of the Compiler	36
6.1 Compiler Parameters	39
6.2 Symbol Table Complex	42
6.2.1 Name Recognition	43
6.2.2 Type Analysis	44
6.2.3 Level/Configuration Table	50
6.2.4 Configuration Table	51
6.2.5 Type Table	51
6.3 Lexical Analysis	52
6.4 Syntactic Analysis	53
6.5 Semantic Analysis	56
7.0 Conclusions	58
References	62

LIBRARY OF L. URBANA-CHAMPAIGN

1.0 Introduction

Programming languages come in two basic flavours--the easy to use, easy to write variety; and the more complicated, but more powerful variety. Both have important applications and, as evidenced by sheer numbers, are quite viable. The former type, as exemplified by BASIC, is adequate for the "simple" program where intricate data structuring is not of primary concern, or for the novice programmer. The latter, typified by PL/I, allow more complex data bases and more powerful manipulations. It becomes the task of the programmer to select the language suitable for the particular project.

CLEOPATRA (Comprehensive Language for Elegant OPERating system and TRANslator design) is a member of the latter class. CLEOPATRA can be characterized by a rather detailed program text, a very modular logical structure, and powerful manipulative abilities. The language specifications as well as a discussion of the implications have been presented by Axel T. Schreiner in [1] and [2]. This thesis presents the initial implementation of the analysis pass of a CLEOPATRA compiler. (Certain extensions and restrictions have been imposed on the original language.)

The obvious question that arises when a new language is presented is, "Why another language?". This is indeed a valid question, and a language, to be a worthy and viable tool, must be able to address this question. In the present case, the objectives have been to produce a compiler for a language that:

- 1) Allows user-defined data types;
- 2) Can act as a laboratory for new features such as:
 - Decision tables, and
 - Powerful control structures that facilitate the design of programs.

Within this framework, the user is allowed a certain amount of flexibility to produce a well structured and concise program for compilation. Further, in an effort to produce a "secure" program, full type checking is enforced. For the compiler, a certain level of complexity is introduced since opening and closing of "blocks" or levels of scope may be more frequent. It is also necessary to maintain more complex data bases to provide these special services.

2.0 Accessibility and Scope

In days of old (relatively speaking!), a program "owned" the computer for the duration of its execution. As a result, the computer and all of its facilities became a servant to the user program. Nowadays, of course, the converse is true: the computer runs the program. This being the case, the program and in particular its contents have a life cycle. In a non-procedural language, the whole program is active for the duration of execution. In the more sophisticated languages (e.g. ALGOL), only portions (often called procedures or blocks) are active at any given time. In CLEOPATRA, this basic unit is the configuration. All scope rules revolve around the configuration. However not all elements of a configuration have identical scopes. The scope of an element is a function of its placement in a local or global block. This relationship between global and local elements will be brought forth after an examination of the basic unit in CLEOPATRA--the configuration.

2.1 The Configuration

The structure of the source program written in any procedure oriented language can be described by a tree in the graph theoretic sense. That is, every procedure has a father; the father of the main procedure might be considered to be the run-time environment. It is from this tree that ALGOL-style languages develop the scope or accessibility rules. CLEOPATRA also uses this tree structure concept. Each node of the resulting program tree is called a configuration. (At this stage, a configuration may be thought of as a procedure in the ALGOL or PASCAL sense. This definition is incomplete but sufficient for the moment.) To the compiler, the underlying concept of the configuration is the configuration's position in the configuration tree. Each node in the tree is given a unique number. The crucial point is that each node in the tree defines the configuration's relation to the program as a whole. To the user, a configuration is denoted by a configuration name.

In most programming languages, the static program tree, corresponding to the configuration tree, is determined and constructed from the physical nesting of the source program. That is, one procedure is a descendant of another if the

former's statements are physically placed within the latter's statements. In CLEOPATRA, the tree is determined by logical nesting rather than physical nesting. This is accomplished via the CLEOPATRA structure block. The structure block defines the accessibility of elements in the tree defined from the present configuration (node the the tree) toward its descendants.

To this point, the term used to describe the program's logical structure has been the static program tree. This is the tree that is constructed at compile-time and represents the true logical nesting of the source program. In contrast to the static program tree is the dynamic run-time tree. The dynamic tree is composed of the physically growing and shrinking program in memory. At run-time, configurations are allocated space in memory for their dynamic variables and required return addresses. This space, which grows upon entry to a configuration and shrinks upon exit, is a constituent of the dynamic program tree. The run-time environment is determined by the code generator, and is described in [3]. (The reader will find a good discussion of this topic as well as an excellent reference to compiler principles in [5] and [6].)

2.2 Local vs. Global Scope

To this point, the static program has been created. In a language such as ALGOL, the scope rules would now be automatically defined. This is not the case in CLEOPATRA. There are two basic scope rules rather than one. These can be explained within the framework of the static program tree, and the context of the current configuration tree. The current configuration tree is defined as the currently active configuration plus all descendants of that configuration in the static program tree. In the case of the initial configuration (similar to the main procedure in PL/I) the current configuration tree is the whole program. In the case of a configuration at a maximum nesting level, the current configuration tree is the configuration itself.

2.2.1 Global Scope

The scope rule for global elements is the same as that of ALGOL. A global element will potentially be active and thus available throughout the current configuration tree. However, a global element can become inactive in two ways,

each producing different results. If a local element of the same name as the global element is declared at a deeper nesting level, the global element becomes known throughout the current configuration tree except in the configuration in which the local element is defined. In the second case, if another global element of the same name as the first global element is defined at a deeper nesting level, the first element is known in its current configuration tree until the configuration in which the second element is declared. At this point, the second global element becomes active for its current configuration tree or until its scope is altered by one of these two situations.

An element is made global by being defined in a global data block, type data block, or by means of a global structure block. The global structure block alters the "globalness" slightly. This alteration will be discussed in due time. For now, it can be stated that the general global scope implies activation throughout the current configuration tree unless the name is redefined.

2.2.2 Local Scope

The concept of the local scope is rather unique to CLEOPATRA. Further, its scope is easy to define. A local element is active only for the configuration in which it is defined. Thus, a local element is not known by any of its descendants.

The purpose of such local scope is for scratch variables, counters, and the like. This scope concept helps contribute to program structure, since scratch variables should not be made global.

An element is made local by declaration in a local data block. The analysis pass detects local and global elements and indicates the type for the code generator. The code generator then allocates the local element upon invocation of its configuration and deallocates upon exit.

3.0 The Subset Language

To this point, the term configuration has been defined as a procedure or a node in the static program tree. A configuration is indeed those things, but in a more precise sense, a configuration is a collection of statements defining an accessibility sequence, the data availability, and the actions to be taken on these data. From this definition, it can be concisely stated from what a configuration is formed. A configuration is composed of a routine block and possibly a combination of a structure and/or a data block. In the context of the previous definition, the following correspondences are formed:

<u>CONCEPTUAL BLOCK</u>	<u>ACTUAL BLOCK</u>	<u>USE</u>
Structural	Structure and global structure	Accessibility sequences
Data	Local data global data	Definition and scope of data
Routine	Procedure and operator block	Executable statements

3.1 Structure Blocks

As alluded to previously, the structure block defines the logical nesting of the program's configurations by constructing the static program tree. The scope of a structure block is the current configuration tree. This implies that procedures are logically nested in the manner of ALGOL procedures.

As indicated, the global structure block modifies slightly the global scope rule. A global structure block is associated with a user-defined data type. The global structure block "pulls" the definition of the type to the same level as the structure defining the user-defined type. That is, a user-defined type is declared in a structure block, which is a node in the static program tree. The global structure block pulls the nesting level in the tree to the same level as the defining configuration.

3.2 Data Blocks

Data blocks are used to define data items available to a configuration. A data block is not required for a configuration if the corresponding configuration does not need to possess any new variables. A configuration does own data items declared global in a predecessor configuration. However, good coding practice favours elimination of scratch global variables. Local data blocks are used to create the scratch variables needed by a configuration. As previously stated, an identifier may be declared in the current

configuration with the same name as a previously declared global identifier. In this case, the current identifier is the active element.

3.3 Routine Blocks

The last type of block is the routine block. To this point, the input has established the symbol table and configuration table with the proper declarations and calling sequences. That having been completed, the executable statements can be parsed. The routine block is the only block which must be present in a configuration. This is the case because a "procedure" may use global data and may not need to possess nested configurations, but must contain executable statements.

Two types of routine blocks exist for this purpose--the procedure block and the operator block. The procedure block is used to describe the user's algorithm in the CLEOPATRA language. The operator block defines the actions to be taken by a user-defined operator.

4.0 Implementation of the Subset

As previously stated, this thesis is the presentation of one aspect of the subset language--the analysis of input through the generation of intermediate text. The purpose of the current implementation is not only to produce a working compiler, but also to determine the types of algorithms required to implement the language constructs.

The subset has been defined in [4]. The subset specifications will not be repeated here except in those areas where changes have been made. Though the code generator was complete before the analysis pass was begun, changes are still possible so long as the code generator receives the required input. In some cases, restrictions have been placed on input, but in other cases extensions have been added.

4.1 Implementation Rules for Structure Blocks

In [4], all configuration names were required to be unique. This was necessary for the linkage editor being used. This restriction has been removed by the analysis

pass. The analysis pass requires uniqueness within any given level in the configuration tree; however, duplication is allowed between levels. This extension was made so that the user can apply the same scope rules to all names presented for compilation. In fact, the analysis pass forms the unique names for the user and passes them to the code generator. This is done by concatenating "CLEO" with a three digit configuration number (which, as will be recalled, is unique). There are two cases where this transformation does not occur. These are for the procedures "INPUT" and "OUTPUT". The code generator uses the input and output routines supplied by the linkage editor. It is, therefore, imperative that the linkage editor actually gets the names "INPUT" and "OUTPUT". This conversion mechanism also allows another extension. Formerly, configuration names were restricted to seven characters. By the above method, there is no restriction in length of these names. The name table constructed by the analysis pass indicates both the configuration name given by the user, and the configuration name passed to the code generator. It should also be noted that the linkage editor map will show the converted names (i.e. the unique names constructed by the analysis pass) and not the user's configuration names. The correspondence can be found by using the name table which is a default option of the analysis pass.

In constructing the static program tree, the analysis pass accepts the first configuration name as the root of the tree (i.e. as the main program in the PL/I sense). This first configuration presented for compilation then is not predefined, but after that point the declare-before-use rule is strictly enforced for structure blocks. The implication is that no configuration may be presented before it is given in a structure block. This is not solely to enforce the declaration rule: the analysis pass has no way of determining where a configuration fits into the configuration tree if its nesting has not been declared along with that of its predecessors. This does not contrast with other languages since physical nesting implies the structure in other block structured languages. An important distinction must be made: A structure block "Y" does not define configuration "Y". Rather, some structure block "X" defines "Y" where "X" is the ancestor of "Y"--i.e. "X" owns "Y".

4.2 Implementation Rules for Data Blocks

The subset language supports local, global and type_data. The analysis pass requires that all identifiers must be declared before their use. Further, the analysis pass provides complete scope and type checking.

In [4], identifiers had been limited to 31 characters in length. The analysis pass puts no restriction on the length of identifiers.

The subset supports four basic types of data along with user-defined types. The four basic types are: 1) Bit; 2) Character; 3) Integer; and 4) Long_integer. The subset does not support the type pointer. A discussion of the problems encountered attempting to implement pointers can be found in Chapter 5. Real, decimal, and octal are not supported by the code generator and thus not supported by the analysis pass. Arrays are permitted with the restrictions as noted in [4].

An important attribute of CLEOPATRA is its support of user-defined data types. There is not a single specific block for this definition; however, the construct type-pack exists for this purpose. A type-pack is a group of the basic blocks which, when combined, define the structure of

the user-defined type as well as configurations which may access the data type. The name given the user-defined type is defined in a structure block to assign to it a position in the configuration tree. Next, a global data block is used to define the underlying representation of the type. It should be noted that the code generator places certain restrictions on the contents of a user-defined type. These restrictions exist in the subset language and are enforced by the analysis pass. The last element of the type-pack is the global structure block, which was mentioned earlier. Local data and local structure blocks may also be applied to a user-defined type.

One other type of data exists and it is the literal, or self-defining constant (for example: 12 or the character string C.abc). These data do not require the define-before-use rule. Several extensions of the language have been made to allow more flexibility for the user. Blanks are allowed between the period and the digits in the hexadecimal, binary, and long_integer strings. (for example: F. 20 is a legal long_integer representation for 20.) Character strings are limited to 256 characters. This restriction is required by the code generator and is enforced by the analysis pass. (Characters after the first 256 are truncated.) Furthermore, the analysis pass supports a full upper and lower case alphabetic character set.

As stated in [1], [2], and [4], CLEOPATRA does not allow automatic data type conversion. There are, however, various builtin functions for this purpose, as well as the ability to define operators.

Storage management is handled in three ways: constant, deferred, and automatic. The mechanism for this handling is treated in detail in [3]. Arrays may not have dynamic bounds. Most of the array partitioning and handling facilities given in the full language are not available in this subset (see [4]).

4.3 Implementation Rules for Routine Blocks

The routine block contains the actual executable statements for the exposition of the specific algorithm to be implemented. The routine block is composed of any number of statements which operate on data "owned" by the configuration and any global data defined previously in the tree. The format and details of the statements will be presented shortly.

There are two types of routine blocks: operator and procedure. The operator block is used to define the action of a user-defined operator on its operand(s). In general, the main algorithm that the user is implementing will be coded in a procedure block.

4.4 Statements

Routine blocks are composed of two basic types of statements. These are: 1) simple, and 2) compound. Simple statements include the following:

- a) Expression
- b) IF ... THEN ... ELSE
- c) RETURN
- d) EXIT

The compound statement combines the simple statements into a more powerful control statement. These are:

- a) BEGIN ... END for bracketing statements.

- b) ITERATE, with options FOR, WHILE, WHEN and EXIT.
- c) DECISION which allows selective execution of statements.

A key difference between CLEOPATRA and most other languages is that expressions are evaluated from right to left. (This is also the method employed in APL) Further, there is no operator precedence (parenthesization forces precedence for the parenthesized quantity). CLEOPATRA allows the user to define operators as does ALGOL W. However, ALGOL W maintains operator precedence by requiring the user to assign a precedence number to each user-defined operator.

User-defined operators may also have parameter lists similar to those of a procedure except that the parameter lists may exist on the right and/or the left side of the operator. In fact, the code generator only accepts right-side parameters. Therefore, the analysis pass converts all left parameters into right parameters. The syntax for parameter lists to operators has been changed to remove an ambiguity in the use of operators with parameters. Formerly, parameters to operators were denoted by placing the parameters in parentheses. The problem arose that if a

binary and unary operator, each with parameters, were placed next to each other, the analysis pass would not be able to resolve the question of which parameter list belonged to which operator. This is because of the ambiguity and the fact that the operators in question may have unary and binary components. Thus parameters to operators must be denoted by placing an apostrophe (') on the side of the parameter list closest to the operator.

The last restriction on expressions is that there may be no more than 50 elements in an expression. This is because the code generator places a limit on the length of an expression in intermediate text form.

One change has been made to the language specifications of the FOR statement. Previously [4], the following would be a legal albeit meaningless statement:

```
FOR identifier ; ; ;
```

Although most programmers would refrain from such a nebulous statement, the analysis pass would have to recognize the construction and the code generator would have to generate code for it. The meaning is certainly questionable, as would be the resultant code. Therefore, the following FOR statement has been implemented:

```

FOR Statement ::= FOR identifier [FROM expression]
                STEP expression
                [[:] UPTO | DOWNTO expression ] ;

```

In the absence of the FROM clause, the initial value of the identifier used as the index is its present value. If the UPTO DOWNTO expression option is not used, the effect is to increment the index and continue the loop indefinitely. This should only be used in conjunction with a WHEN, WHILE, or EXIT statement. This form of the FOR statement allows as flexible and powerful implementation as the code generator via the intermediate text will accept.

The first proposed revision was similar to the version implemented, however it would have made optional the STEP expression phrase when using UPTO or DOWNTO expression.

```

FOR Statement ::= FOR identifier [FROM expression]
                [ STEP expression ]
                [[:] UPTO | DOWNTO expression ] ;

```

The default when STEP is omitted would be one. This is the

optimal solution, but because of the requirements of the intermediate text, it was not possible to implement this form.

All keywords in the subset have been reserved. A list of these reserved words along with their symbol table entry numbers can be found in Figure 1. Figure 2 gives a list of the builtin operators provided by the subset. These may be redefined in an operator block and thus are not reserved. The code generator requires that there be no more than 1500 symbol table entries in addition to the list of reserved words and operators given. Although 94 keywords are given in the figures, there are actually 122 keywords. The discrepancy lies in the fact that most of the operators have more than one entry (e.g. 66 and 69). The operators, though apparently equivalent, are not because they operate on different operand types. The negative sign in entry 66 is a unary negative and operates on integers. The negative sign in entry 69 is a binary negative and operates on integers. This same situation applies to most operators. The effect is invisible to the user, however, since a semantic analysis routine is called during the parsing of expressions to determine the semantically correct operator.

<u>Entry</u> <u>Number</u>	<u>Symbol</u>	<u>Entry</u> <u>Number</u>	<u>Symbol</u>
1	ACTION	2	ALLOCATE
3	BEGIN	4	DECISION
5	DOWNT0	6	ELSE
7	END	8	EXIT
9	FOR	10	FROM
11	IF	12	ITERATE
13	NIL	14	OPERATOR
15	PROCEDURE	16	RELEASE
17	RETURN	18	STEP
19	THEN	20	WHEN
21	UPT0	22	WHILE
23	ADDRESS	24	ALIAS
25	B	26	BIT
27	BUILT	28	BY
29	C	30	CHARACTER
31	COMMENT	32	COMPILE

Figure 1a

Reserved Words

<u>Entry</u> <u>Number</u>	<u>Symbol</u>	<u>Entry</u> <u>Number</u>	<u>Symbol</u>
33	CONSTANT	34	DATA
35	DEFER	36	EXTENTS
37	F	38	GLOBAL
39	IN	40	INIT
41	INTEGER	42	INTO
43	LONGINTEGER	44	POINTER
45	RETURNS	46	RIGHT
47	S	48	TO
49	VECTOR	50	TYPE
51	X	52	STRUCTURE
53	FALSE	54	FIRST
55	LARGE	56	LAST
57	SMALL	58	TRUE

Figure 1b

Reserved Words (continued)

<u>Entry</u> <u>Number</u>	<u>Symbol</u>	<u>Entry</u> <u>Number</u>	<u>Symbol</u>
66	-	67	ABS
68	+	69	-
70	*	71	//
72	**	73	MOD
74	AND	75	LBOUND
76	OR	77	~
78	==	79	=
80	>	81	<
82	>=	83	<=
84	:=	85	<UNUSED>
86	LENGTH	87	HBOUND
88	->	89	<-
90	"->	91	?->
92	"	93	LINT
94	CHAR		

Figure 2

Built-in Operators

5.0 Further Considerations

During the implementation of the analysis pass, several problems arose. In most cases, by some modification, they were overcome. However, two major problems have persisted. In both cases, implementation was attempted within the constraints of the language, facilities, and code generator. Unfortunately, the attempts met with limited success. Valuable information was gleaned from this process and is presented below so that future efforts might profit from the results thus far.

5.1 Pointer Variables

A major dilemma arose in the implementation of pointer variables. The BNF (Backus Naur Form) of the productions causing the problem is:

```
Basic_Ref_Type ::= ... Pointer(ref_type)
```

```
Ref_Type ::= Basic_Ref_Type ...
```

The quandary then became the following:

At data definition time, the compiler does not know the name of the variables pointed to and so must store the type pointed to as pointer, as pointer.... However, the symbol table does not allow control over the level of pointers nor the final target of the pointer. Therefore the strong typing could easily be circumvented, and nothing in the user's region would be safe.

The problem becomes complicated by the fact that the symbol table, which was designed before the analysis pass was begun, does not inherently support indirect pointers. One advantage of CLEOPATRA is that all types are checked in the analysis pass for compatibility of source and target. It is therefore imperative to have the facilities to deduce and retain types for identifiers. Type checking is indeed a two level problem in CLEOPATRA. At the point of definition of the variable, all that is known is that it is a pointer to a pointer to a pointer.... Later, in the usage context the name of the variable pointed to, pointed to... must be checked. Thus the levels of offset must be known.

Several solutions were hypothesized and each summarily discarded save for the last. Among the proposals entertained were:

- 1) Restricting pointers to point to basic types only, thus eliminating multi-level pointers. This appeared more restrictive than valuable.
- 2) Constructing a symbol table for the analysis pass, then reformatting before passing to the code generator. This method would be wasteful of space and especially of time.
- 3) Removing pointers from the subset.

Since the current implementation is a subset and an investigation into the methods for implementation of CLEOPATRA, the removal appeared to be the best solution. Indeed, it has been argued that pointers should not be included in programming languages [7].

Though they have been removed, and irrespective of the debate, pointers are in the full language and must eventually be handled. After considerable effort in an attempt to implement them, an adequate solution has been formulated for use in the full language. Rather than a "type returned/pointed to" field in the type analysis table

of the Symbol Table Complex, a `ref_type` field is suggested. This would be a pointer to a linked list containing the type pointed to and a pointer to the next level of offset if necessary. This would allow full implementation, compact size since the nodes could be allocated dynamically, and simple yet complete type checking.

5.2 Presentation of Blocks for Compilation

In the CLEOPATRA language description presented in [2], blocks could be presented in almost any order. In [4], blocks must be grouped by configuration with the routine block following the structure and data blocks. These two orderings imply a very different analysis pass.

As will be recalled, an important function performed by the analysis pass is the construction of the static program tree. This activity is mostly invisible to the code generator since it presumes proper (or repaired) input. The analysis pass, on the other hand, bases all of its activities around the knowledge of this tree. In the first ordering (actually non-ordering) the tree may not be completed to the proper level when a routine block for a

heretofore undeclared configuration arrives. Several alternatives present themselves at this point: 1) Declare the new configuration to be a descendant of the current configuration; 2) Flush the block; and 3) Take two passes over the input. We shall discuss the three in order.

By declaring that the routine block for an unknown configuration automatically becomes a direct descendant of the current block the initial question of where to place the configuration in the tree is resolved. However, this immediately presents several new problems with possible disastrous results. First, the static tree would have to be "pulled apart" and the new configuration inserted. The physical action of changing the links is not a problem. A larger problem is that the scope of variables and procedures would change. By inserting this configuration, new procedures and variables could be brought into the current scope. This would potentially change variables, present naming conflicts, and alter calling sequences. Without elaborating further, it should be clear that this method is completely infeasible.

The second alternative, to flush the block, would certainly alleviate some of the problems associated with the first alternative. No new blocks or variables would be pulled into a potentially incorrect place in the

configuration tree. Obviously, execution would have to be suppressed since the routine would not be used. Further, any calls on this routine would be illegal. Also, any procedures nested within the routine would be flushed since they would automatically be out of order. Thus, this solution has a very severe snowball effect.

The last alternative is to run a two pass analysis. The first pass would collect all elements and sort out the structure of the static tree. Then the second pass would resolve scopes based on the static tree. This method would solve the problems presented above. The disadvantages are that the whole compiler would require three passes (two for analysis and one for code generation), and some form of the input would have to be retained for the analysis pass. The important point is that the problem of the static tree would be solved at the expense of additional time and space.

The requirement that blocks must be presented by configuration with routine blocks last alleviates the problem above and requires only one pass for analysis. The problem here is that there is slightly less generality in the presentation of blocks for compilation.

The actual implementation is somewhat of a compromise between all the methods. The analysis pass was implemented as a single pass implying a certain amount of ordering. We

require a configuration to be defined in a structure block before any other blocks for that configuration are presented (except for the initial configuration). Next in order of presentation must be any type_data blocks where the name of the type_data block was declared in the immediately preceeding structure block. This is necessary to preserve the list of like named predecessors. When a block is closed, a predecessor for each element being closed is activated if it exists. In order to keep type_data open this ordering must be preserved. Lastly, the data blocks and routine blocks are presented in the following manner. Any data block must precede its routine block but, need not immediately precede it. These blocks must be presented in level order--just as the scope rules. The most important rule is that a structure block must precede all other blocks in its configuration. However, all structure blocks do not have to be grouped together.

A considerable amount of time was spent attempting to find a better compromise. Indeed, at one point all activities ground to a halt while efforts were turned to the opening and closing algorithm. Finally, the above solution was chosen. Along the way, a very promising algorithm was in part derived. The concept is presented here for further consideration and refinement.

This method employs a 6 x 3 bit map for each configuration. The format is: `types_of_blocks x legal_use`. Precisely, the form is:

	Allow	May Define	Has been Defined
Routine	.	.	.
Global Data	.	.	.
Type_Data	.	.	.
Local Data	.	.	.
Global Structure	.	.	.
Local Structure	.	.	.

The "allow" column is a function of the type of configuration--algorithm or type_pack. The "may define" is initialized to "yes" and switched to "no" later as a function of the "has been defined" entry. The "has been defined" is initialized to "no" and changes as compilation proceeds.

As an example of the use of the bit map, consider the following situations:

- 1) When defining a routine block (row 1), we know that from this point on, we can not present data blocks for this configuration. So, all data block bits are set to "may not define".
- 2) When defining a global data block for the current configuration, we can not define type data or global data for predecessors in the configuration tree. Therefore we climb up the configuration tree setting the "may not define" bits.
- 3) When defining a local structure block for a configuration, we can not define structure blocks for any predecessor configurations. So, the "may not define" bit is set for all predecessors.

Upon entering a block, the bit map is checked to determine whether the block just entered is in a legal position.

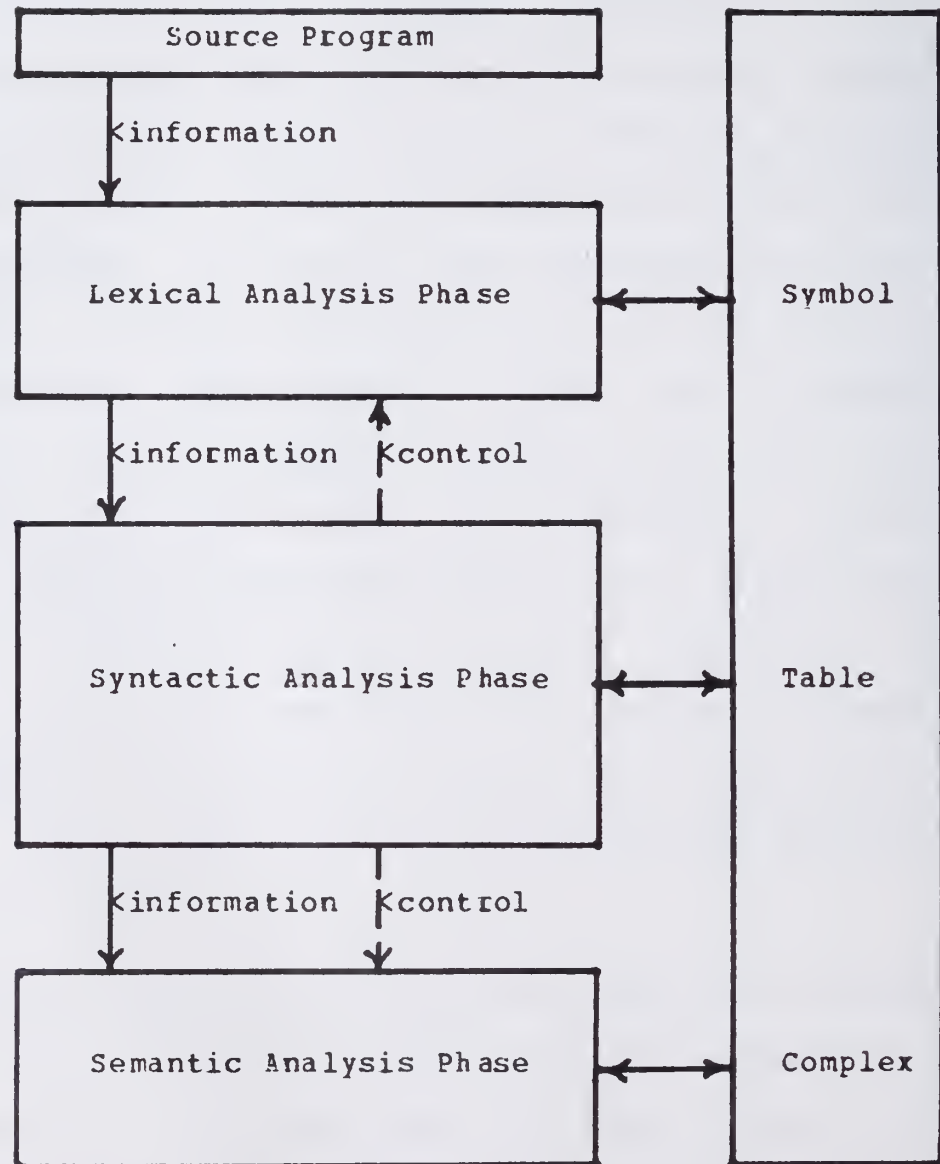
This method would require little overhead and only the 18-bit map. The method has not been proved for all cases but because of the speed and small space requirements, it seems to be the best alternative method.

6.0 Structure of the Compiler

The CLEOPATRA compiler has been implemented in two passes. That is, the input or a transformation is examined in two distinct time frames. The first pass--the Analysis Pass--is the interface between the user's source program and the code generator. The analysis pass constructs the static program tree, examines the input to be sure that it is in proper syntactic and semantic form, and produces intermediate text--a hybrid transformation of the source--which is passed to the code generator. If any errors are detected during this analysis, the analysis pass sets the user condition code to 12 which suppresses the invocation of the code generator. If no errors are found, the code generator is called to convert the intermediate text into object code. The details of the code generator can be found in [3].

The analysis pass may be broken into three basic phases (see Figure 3): Lexical Analysis, Syntactic Analysis, and Semantic Analysis. Though the pass has been broken into phases, it must be realized that only the analysis pass is a discrete entity. The analysis pass can be defined as the

interaction of the three phases with the symbol table complex on the source program to produce an internal representation of the user program.



Solid line -- Flow of information
 Broken line -- Flow of control

Figure 3
 Analysis Pass

In the pages that follow, the phases and the symbol table complex will be dissected and discussed as to their function in the compilation process.

6.1 Compiler Parameters

During the design of the analysis pass, several compile-time parameters were incorporated into the code. In all cases, values for the parameters may be passed to the compiler in the parameter field of the job control language. If no parameters are passed, the default values are used. In general, the default parameters will be sufficient for a compilation. A listing of all options and their value for the current compilation is printed on a page before the source listing. These parameters are listed in Figure 4 and are encoded by prefixing the name of the parameter with "DEBUG." followed by the parameter followed by "=value_to_be_used,". In the case of the last parameter, the comma is replaced by a semicolon.

<u>Parameter</u>	<u>Action</u>	<u>Default</u>
BLK_DUMP	Output block tables.	Yes
CONFIG_DUMP	Output configuration tables.	Yes
DEBUG_CRD	Output a trace of the compiler along with selected values beginning at this card number. To be used only on compiler error as it generates considerable output.	30000
DEBUG_PROC	Same as above. It may be enabled either initially or by the above card number.	No

Figure 4a

Compiler Parameters

<u>Parameter</u>	<u>Action</u>	<u>Default</u>
INT_TXT	Output a listing of the intermediate text--this requires a "//GO.ITEXT DD SYSOUT=A" card if the cataloged procedure is not used.	No
LINECT	Number of source cards to be listed on a page.	58
NAM_TAB_DUMP	Output the name table.	Yes
OPEN_TST	Output block tables, configuration tables, and selected values upon entering a new block.	No
SOURCE	Output a source listing.	Yes
SYM_TAB_DUMP	Output the symbol table.	Yes
TYP_TAB_DUMP	Output the type tables.	Yes

Figure 4b

Compiler Parameters (continued)

Since many large tables are required by the analysis pass, compilation began to require large amounts of main memory. Therefore, the analysis pass has been overlayed to reduce this requirement. The overhead in swapping the overlays is minimal compared to the saving in space.

The analysis pass was originally designed to have expandable tables (symbol table, block table etc.) for increased generality. Unfortunately, this feature had to be removed. As the amount of code increased, the cost of recompilation of the analysis pass increased prohibitively. Therefore, other methods (external procedures in PL/I, and use of load modules) which do not allow dynamic bounds on arrays were employed. This saved in the cost of implementation, but forced the use of static bounded tables. In fact, this is not too limiting since the code generator has a fixed bound on the size of the symbol table.

6.2 Symbol Table Complex

The Symbol Table Complex is the major data base for the compiler. This is due not only to its information content, but also to its physical size and time spent in its

manipulations. This size is in part due to the power of the compiler. The symbol table complex is used by all phases although not all phases may change an entry. The symbol table complex is composed of four basic modules:

- 1) Name Recognition
- 2) Type Analysis
- 3) Level / Configuration Table
- 4) Configuration table
- 5) Type Table

In the following sections each table will be presented for an overview of its function.

6.2.1 Name Recognition

The Name Recognition Table contains the actual character representation of the input tokens. This table consists of a linear string of all tokens concatenated together. The table also contains a vector of three pointers for each symbol table entry. The first is a pointer to the start of the token in the string. The second

is a length count. The last is a link of the like-names (like but not equivalent). As previously mentioned, two similarly named symbols may exist provided they have different scopes. However, only one entry is kept in the name table in order to save space.

Name recognition is the first action taken by the symbol table manager in searching for an entry. The search technique employed is a hash table with chaining for duplicates. After some experimentation a suitable hash function was chosen and mapped into 256 hashing buckets. When two different tokens hash to the same bucket, the first is inserted into the table in the usual fashion, but the second is chained by a link from the first.

6.2.2 Type Analysis

The Type Analysis table contains the type representation and parts of the scope values for symbols. As discussed above, the lexical analysis phase inserts values into the type analysis table. The fields and their values are given below:

<u>Field</u>	<u>Function</u>
Type	Type of symbol or returned type: <ul style="list-style-type: none"> 1 .. Long_integer 2 .. Character 3 .. Error 7 .. Pointer 8 .. Bit 9 .. User Defined 13 .. Label 15 .. Integer.
Constant	Set if constant data.
Array	Set if array.
Literal	Set if self-defining constant.
Initial	Set if symbol is initialized.
Local	Set if local symbol.
Defer	Set if deferred storage.
Formal	Set if formal parameter.
Alias	Set if alias name.
In_type	Set if part of user-defined type.

Link	Set if link item (e.g. configuration name).
Entry	Set if an entry item. following types: 01 .. if procedure 10 .. if unary operator 11 .. if binary operator.
Ptr	Type of symbol pointed to (not implemented).
Aptr	Set if points to array (not implemented).
Unused	Analysis pass sets bit number 1 if the identifier is a data item, and 2 if it is a read-only data item.
Blk	Configuration number of surrounding configuration.

Blklevel	Nesting level of the configuration if the item is a configuration name, or the nesting level of the surrounding configuration if it is not a configuration.
----------	--

Atr1	Depends on the item.
------	----------------------

Atr2	Depends on the item.
------	----------------------

Most of the entries are self-explanatory from the context of use. Those not clear are expanded upon below.

<u>Entry Type</u>	<u>Special Field Usage</u>
-------------------	----------------------------

Configuration name	Reserve two symbol table rows. Set type returned in TYPE of row two. Link bit is set in rows one and two. Entry bit is set in row two. Atr1 in row one contains the configuration number.
--------------------	---

Atr2 in row one points to the configuration name in the constant table.

Atr1 in row two contains the number of parameters to the procedure/operator.

Atr2 in row two contains a pointer to the position of the entry pointed to if the procedure returns a pointer. (not used)

Data item

Atr1 is the maximum length if the item is of type character. Otherwise Atr1 is the configuration number of the surrounding configuration.

Atr2 points to the row of the value to which the data item is initialized if the item has the initial attribute.

Character

Length in Atr1.

If initialized, Atr2 points to the initial value in the constant table.

Bit,	
Integer,	
Long_integer,	
Constant	The value is in Atr2 unless long_integer, in which case the value is overlayed in Atr1 and Atr2.
Alias	Set alias bit and link if necessary. Atr2 points to the major name.
User-defined types	Blk set to zero for elements and in_type bit is set.

The type analysis table is one of two tables passed to the code generator. The other is the constant table. The constant table contains the following entries:

- 1) Seven character unique configuration names as described previously.
- 2) The number of extents and bounds for arrays.
- 3) The value of character literals.

Both tables are passed to the code generator in the intermediate text file. On entry to a routine block, the type analysis table entries from the last element transmitted through to the current top of the table and the whole constant table are placed in the intermediate text file. (The last element of the type analysis table is denoted by eleven ones in the unused field.) This is followed by the intermediate text for the routine.

6.2.3 Level/Configuration Table

The Level/Configuration table contains the information about the activation of symbols. One element of the table is a field that links together all symbols of the same configuration. This field is used for activation of symbols upon entry to a new block. Other fields are used to form a chain of entries and their predecessors.

6.2.4 Configuration Table

The Configuration Table is a presentation of the static program tree. It consists of a list of configuration numbers along with their immediate predecessors. There is also a pointer to a symbol number in the level/configuration table which thus links all elements of the same configuration. This table is built mainly from the structure blocks.

6.2.5 Type Table

The Type Table holds the attributes of parameters to operators and procedures. When operators and procedures are declared, only the type of the parameter is given. It is at the point when the operator or procedure is called that the actual identifier is found. Parameters can be of any available type, arrays, and left or right in the case of operators. There is no specific bit in the type table to denote an array entry. This is done indirectly by recording the number of extents for the entry. The number of extents is always zero unless the item is an array in which case it

has a positive value. The semantic processor requires this information in type checking.

6.3 Lexical Analysis

As shown in Figure 3, the Lexical Analysis phase is the first phase encountered by the input. The Analysis pass occurs as one pass thus there is an interaction among the phases. The lexical analysis phase performs the most rudimentary albeit important work on the input. First, the input stream is tokenized by the productions in the language specifications. Comments and blanks are eliminated.

(See Figure 3.)

The scanner is able to recognize certain types and resolve them--bit values, integers, long_integers. In the case where the scanner is able to determine the type (due to the syntax) the type may be inserted into the symbol table. When the scanner is not able to make this determination, it still is able to reduce the possibilities. It thus inserts an unresolved type.

The scanner calls upon a routine of the Symbol Table Manager to look up the symbol in the symbol table. Depending on the context, another routine can be called to insert the token into the Symbol Table Complex. Other utilities include reading routines and a routine to convert radixes since CLEOPATRA accepts input in binary, decimal, or hexadecimal integers. There is also a routine to convert characters to their integer representation, and integers to their character representation. The Lexical Analysis phase is driven by the Syntax Analyzer. The Syntax Analyzer "knows" what to look for and the context in which it resides. It thus knows whether it is in a declarative context or merely looking for a defined token.

6.4 Syntactic Analysis

If one phase of the compiler could be considered the driver for the analysis pass, the Syntactic Analysis phase could qualify. It is the syntax of the language upon which the parse is based.

The syntax analyzer is broken (and overlayed) into five major modules. These modules follow the same lines as the block structure: Local Data, Global Data, Local Structure, Global Structure, and Type Data. The basic flow of the syntax analyzer is: While source text exists, the main program determines the type of block that is being presented, and calls the proper block processor. The block processors are not completely self-contained; they all share certain modules such as the error handler. However, it is clear that once parsing begins on one type of block, the other block parsers are unnecessary. The selected processor then continues parsing until the block terminates. At that point, control returns to the main program for selection of the next block.

Several different parsing methods were considered for the current implementation. Upon examination of the BNF specifications, it was found that in almost all cases, the parsing machinery could determine "where it was" in the parse by looking at the current symbol. In the worst case, one symbol look-ahead was necessary. For this reason, a recursive descent parser was chosen for all elements in the language. Although this is not as fast as an LR parser, implementation was quicker and did not require the calculation of the parsing tables.

Error correction is handled on a "need to know" basis. The parser continues parsing after an error is encountered until it is unable to determine where to continue in the parse. When this occurs, the parser flushes the input until it is able to determine where to continue. In the worst case, this means flushing the current statement which may be a compound statement (e.g. a FOR loop). In other cases only part of a statement is flushed. In any case, one or more error messages will be output. If a compiler error occurs, at least one error message is printed (which would likely be the cause of the severe error), and all tables are printed along with selected compiler variable values.

An important function of the syntax analyzer is to insert values into the symbol table complex. It is at this point that the types of symbols are resolved. Recall that the lexical analysis phase inserted a type which was usually unresolved. The syntax analyzer is able to determine the types that the lexical analyzer is not able to distinguish.

6.5 Semantic Analysis

It has been stated on several occasions that CLEOPATRA is a very type conscious language. That is, mixed mode operations are not allowed except by user-defined operators designed for this purpose. It is the task of the semantic processing modules to monitor all types and their usage.

The primary semantic processor is associated with the expression parser. Each operand is pushed onto a type stack. As the associated operator arrives, the type of the top entry (or entries for binary operators) is compared to the type of the operator. If the types match, the parse continues. If the types are unmatched, the semantic processor searches the symbol table via the configuration link to find another entry for the operator which has the proper types. If the proper type is not found, an error message is emitted and the call to the code generator is inhibited by setting the condition code.

Another important aspect of semantics in CLEOPATRA is the analysis of parameter lists. The full connection is a three-way association: The declaration of a procedure or operator specifies the attributes of the parameters. The data block for the respective block must declare a variable of each of the attributes to serve as a target for the

parameter. At this point no parameter checking is done, however, since the positional association has not yet been made. The third component of the association ties the package together. In the definition of the routine block, a `name_list` is specified if there are parameters to the routine. At this point identifiers must be placed positionally as to their type and the type in the declaration of the procedure or operator. A semantic processor then compares the position of the parameter to the expected type at that position against the type of the identifier. If there is a mismatch--too few or too many parameters--an error message is generated.

7.0 Conclusions

The objective of the current research has been to implement a parser and intermediate text generator for the CLEOPATRA subset language. Further, this implementation was to serve as a test of the feasibility of the language itself. It was to serve as an analysis of the algorithms and data bases required to provide the facilities of the CLEOPATRA language. In general, the research has been successful in this regard. Certain limitations were placed on the full language, but every attempt was made to keep the language intact.

Is the subset implementation the perfect language? Certainly we would like to respond affirmatively to that question but, in fact, the answer is "not really". All of the major control structures and blocks have been implemented. But, other important features have been omitted. Some of these omissions include (recalling from Chapter 5):

- 1) Pointers;
- 2) Complete freedom in the order of presentation of blocks for compilation;
- 3) Allowing arrays within a user-defined type (see [4]).

Notwithstanding these omissions, the language CLEOPATRA and the present subset are viable tools in the programmer's repertoire. During the coding of the analysis pass, the facilities of CLEOPATRA would have made implementation much easier and cleaner. Further, the language leads quite naturally to a well structured and clean program. The data bases used in the subset should be sufficient for later implementations. Indeed, as a test for feasibility, the present implementation has demonstrated that the language is feasible. The present research has indicated a course of action for further efforts on the implementation of the full language.

Initially, a study of the use, need and desirability of the basic type POINTER should be undertaken. As stated in Chapter 5, it has been argued that pointers are an artifact from compilers gone by. A careful analysis of whether pointers should be available and what should be a legal target for a pointer if they are to be allowed should be done first. If the type POINTER is to be retained, the method of implementation should be examined. This would answer such questions as: 1) "How is type checking to be handled?"; and 2) "How can arrays of pointers be implemented?".

The next area of study should be in symbol table design. In Chapter 5, it was stated that the present symbol table is insufficient for indirect pointers. This problem, along with some general restructuring, should be undertaken--possibly in conjunction with the study of pointers. (Suggestions have been given in Chapter 5.)

The presentation order of blocks should be examined relative to the bit map discussed in Chapter 5. As stated, the bit map seems to be the "cheapest" method to allow maximum flexibility in presentation order, yet retaining the general two pass method. This problem is somewhat less crucial since placing the restriction used in the subset does not appear too restrictive. However, the algorithm should not be too difficult if the bit map is true for all cases.

The last major area to be examined before implementing the full language is a study of optimal parsing and code generation methods for the CLEOPATRA language. A table driven method might be considered, although table size might make such a method impractical. In all considerations, the design of a well structured compiler with the capability of being overlayed should be paramount.

The above analyses having been completed, the full CLEOPATRA language could be implemented. With the lessons of the subset and the suggested analyses, CLEOPATRA would indeed be a very effective programming language.

References

- [1] Schreiner, Axel T., "A Proposal for Another System Implementation Language", Ph.D. Thesis, Department of Computer Science, University of Illinois, Urbana, Illinois, 1974.
- [2] Schreiner, Axel T., "Comprehensive Language for Elegant Operating System and Translator Design", Technical Report UIUCDCS-R-74-646, Department of Computer Science, University of Illinois, Urbana, Illinois, 1974.
- [3] Halbur, John D., "A Code Generator for the CLEOPATRA Language", Masters Thesis, Department of Computer Science, University of Illinois, Urbana, Illinois, 1975.
- [4] Halbur John D., "CLEOPATRA Code Generator User's Guide", Technical Report UIUCDCS-R-76-740, Department of Computer Science, University of Illinois, Urbana, Illinois, 1976.
- [5] Gries, David, Compiler Construction for Digital Computers, John Wiley and Sons, Inc., New York 1971.
- [6] Baur, F. L., Eckel, J., Compiler Construction--An Advanced Course, Springer-Verlag, New York 1974.
- [7] Hoare, C.A.R., Notes on Data Structuring, in Dahl, Dijkstra, and Hoare, Structured Programming, Academic Press, New York, pp. 83-174, 1972.

BIBLIOGRAPHIC DATA SHEET	1. Report No. UIUCDCS-R-76-834	2.	3. Recipient's Accession No.
	4. Title and Subtitle IMPLEMENTATION OF THE LANGUAGE CLEOPATRA: THE ANALYSIS PASS		5. Report Date October 1976
7. Author(s) Scott Harley Fisher		6.	
9. Performing Organization Name and Address Department of Computer Science University of Illinois at Urbana-Champaign Urbana, Illinois 61801		8. Performing Organization Rept. No.	
		10. Project/Task/Work Unit No.	
		11. Contract/Grant No.	
12. Sponsoring Organization Name and Address Department of Computer Science University of Illinois at Urbana-Champaign Urbana, Illinois 61801		13. Type of Report & Period Covered Master's Thesis	
		14.	
15. Supplementary Notes			
16. Abstracts CLEOPATRA is a general purpose language with features suitable for systems programming. A compiler for the language CLEOPATRA has been implemented in two passes. This report describes the analysis pass which produces an intermediate text suitable for the code generation pass. The analysis pass was written in PL/I for the IBM 360 computer. Due to the facilities of the language, the analysis pass requires innovative data structures and algorithms - these are reported herein.			
17. Key Words and Document Analysis. 17a. Descriptors Block Structured Language Compilation Compilers Intermediate Text Programming Languages Symbol Table Management Parsing			
17b. Identifiers/Open-Ended Terms Systems Implementation Languages CLEOPATRA			
17c. COSATI Field/Group			
18. Availability Statement RELEASE UNLIMITED		19. Security Class (This Report) UNCLASSIFIED	21. No. of Pages 66
		20. Security Class (This Page) UNCLASSIFIED	22. Price

REPORT OF THE URBANA CAMPAIGN

FEB 24 1977

UNIVERSITY OF CALIFORNIA
LIBRARY

THE HISTORY OF THE UNITED STATES

JAN 19 1978



UNIVERSITY OF ILLINOIS-URBANA
510.84 IL6R no. C002 no. 830-835(1976
Implementation of the language CLEOPATRA



3 0112 088403073